

# Performance Analysis of Parallel Systems: Approaches and Open Problems

Daniel A. Reed\*   Ruth A. Aydt   Luiz DeRose   Celso L. Mendes  
Randy L. Ribler   Eric Shaffer   Huseyin Simitci   Jeffrey S. Vetter  
Daniel R. Wells   Shannon Whitmore  
Ying Zhang

Department of Computer Science  
University of Illinois  
Urbana, Illinois 61801 USA

## Abstract

Parallel computing is rapidly evolving to include heterogeneous collections of distributed and parallel systems. Concurrently, applications are becoming increasingly multidisciplinary with code libraries implemented using diverse programming models. To optimize the behavior of complex applications on heterogeneous systems, performance analysis software must also evolve, replacing *post-mortem* analysis with real-time, adaptive optimization, tightly integrating compile-time analysis with performance measurement and prediction, and supporting high-modality visualization and software manipulation. In this paper, we briefly survey the state of the art in each of these areas and sketch a series of open research problems.

## 1 Introduction

As parallel computing evolves from homogeneous parallel systems to distributed collections of heterogeneous systems (i.e., the computational grid), application developers face new and more complex performance tuning and optimization problems. Current users of parallel systems often complain that it is

difficult to achieve a high fraction of the theoretical performance peak — the time varying resources of the computational grid further exacerbate these problems. Moreover, the sensitivity of parallel system performance to slight changes in application code, together with the large number of potential application performance problems (e.g., load balance, data locality, or input/output) and continually evolving system software, make application tuning complex and often counter-intuitive.

At present, we have few, if any reliable techniques for predicting application performance from first principles. Instead, we must exploit experimental techniques, making performance analysis subject to the same constraints as other experimental sciences. Furthermore, performance tools must be simple and intuitive. Unless compelled by circumstances, most users are unwilling to invest great time and effort to learn the syntax and semantics of new performance tools; they often view performance optimization as an unavoidable evil. Hence, portability and ease of use are critical to the acceptance of new performance tools. Simply put, the goal of experimental performance analysis is to provide insight into application behavior and performance bottlenecks by efficiently capturing and intuitively presenting performance data.

The remainder of this paper is organized as follows. In §2, we begin by describing traditional approaches to performance instrumentation and analysis. Following this, in §3 we outline a set of challenges and research directions facing the performance analysis

---

\*This work was supported in part by the Defense Advanced Research Projects Agency under DARPA contracts DABT63-94-C0049 (SIO Initiative), F30602-96-C-0161, and DABT63-96-C-0027 by the National Science Foundation under grants NSF CDA 94-01124 and ASC 97-20202, and by the Department of Energy under contracts DOE B-341494, W-7405-ENG-48, and 1-B-333164.

community, including aggressive compile-time optimization, domain-specific analyses, real-time adaptive performance control and steering, techniques for reducing performance data volume, and performance data immersion and direct manipulation within virtual environments. Finally, §4–§5 discuss related work and summarize our conclusions.

## 2 Measurement and Analysis

Because performance analysis involves measurement, it is prey to the same theoretical and pragmatic pitfalls as other experimental sciences. In particular, it must not unduly perturb the measured system, else the experimental data will not reflect the system’s normal behavior. However, it must be sufficiently detailed to capture the phenomena of interest, and it must relate the measured data to a context that enables the user to optimize performance.

### 2.1 Performance Instrumentation

Historically, performance instrumentation approaches have included counting and sampling, interval timing, and event tracing. Conceptually, each strikes a different balance between instrumentation overhead, data volume, and detail. In addition, each has multiple possible implementation techniques, ranging from completely extrinsic (e.g., an external hardware monitor that counts L2 cache misses via connections to a set of probe points) to completely intrinsic (e.g., inserted code in an application program to compute a histogram of procedure activation lifetimes).

By far the most common method of performance instrumentation is program counter sampling. The widely used UNIX `prof` and `gprof` [11] utilities periodically sample the program counter and compute a histogram of program counter locations. Because the histogram bins correspond to procedure code fragments and sampling occurs at known intervals, histogram bin height is an estimator of the amount of time spent in a particular procedure.

Profiling depends on an external sampling task, leading to coarse granularity and requiring total program execution time to be sufficiently long to accumulate a statistically meaningful sample set. Moreover, profilers often assume a simple mapping from object code to the original source code, which is rarely true on parallel systems with aggressive restructuring compilers; we will return to this topic in §3.1. Finally,

on parallel systems the code execution paths may be different on each processor (e.g., due to data dependent behavior), requiring more complex mappings to application code.

Event counting eliminates some of sampling’s limitations, albeit with additional cost. Because counting is not a statistical measure, the observed frequencies are accurate. However, to obtain timing information, one must periodically timestamp and record the counts.

As the measured analog of sampling, interval timing combines counts with elapsed time measurements. Rather than sampling the program counter periodically to compute the amount of time spent in code fragments, interval timing brackets code fragments with calls to a timing routine.

Unlike counting, which naturally abstracts the occurrence of specific events, or interval timing, which abstracts the frequency of specific events, event tracing generates a complete sequence of events. Hence, event tracing is a more general instrumentation technique than either counting or interval timing; from an event trace, one can compute counts or times — the converse is not true.

The disadvantage of tracing is the potential instrumentation intrusion. Because each event must be timestamped and recorded separately, the potential data volume is large, and the input/output requirements are substantial. Instrumenting procedures to record entry and exit can easily generate 16 KB/second on a single processor if the mean procedure activation lifetime is 500 microseconds and the data associated with each event includes only a four byte event identifier and a four byte timestamp. On a parallel system with hundreds of processors, the data volume can approach 10 MB/second — we will return to this topic and possible solution techniques in §3.2

In general, counting, timing, and tracing occupy different points along the continuum of detail and measurement overhead, and no single instrumentation approach is appropriate in all cases. The choice of a particular approach is dictated by the desired information and the constraints of the underlying instrumentation implementation — some measurements are not feasible in some environments.

### 2.2 Performance Analysis

When large scale parallel systems first emerged, several research groups developed performance visualization systems that displayed the dynamic behavior

of processors and communication networks. Exemplars of this work include Couch’s seminal Seecube system [5], ParaGraph [12], and our own Hyperview and Pablo toolkits [21, 28].

Although widely used to develop applications on early parallel systems, the increasingly complexity of parallel systems exposed several limitations of these toolkits. Perhaps the most obvious constraint was their limited scalability. Performance environment scalability implies not only that the environment must be capable of capturing and analyzing data from very large numbers of processors, but also that it must be capable of presenting the data in ways that are intuitive and instructive.

However, early visualization systems represented the states of individual processors (e.g., by a colored square for each processor) and communication links (e.g., by communication network animations). In consequence, they were limited by workstation screen real estate and do not scale to thousands of processors. In §3.5, we will discuss hierarchical visualization techniques that can accomodate large numbers of processors.

Second, increasing software complexity, high-level programming models (e.g., data parallel and parallel object) and heterogeneous, wide area computing have all exacerbated performance analysis problems. No longer can measured data be related to user code without tight integration of performance measurement systems with compilers and runtime systems [1].

Third, repeated experience has shown that scientific application software developers will eschew powerful, but complex tools in favor of inferior, but easily understood tools. Unless compelled by circumstances, such users are unwilling to invest much time and effort to learn the syntax and semantics of new performance tools; they often view performance optimization as an unavoidable evil. Hence, portability and ease of use are critical to the acceptance of new performance tools.

Fourth, there are at least three classes of potential performance environment users, novice, intermediate, and expert, each with different expectations. Novice users know relatively little about parallel system software or hardware, nor do they wish to learn more than the minimum necessary to optimize the performance of their application codes. In contrast, intermediate users are willing to conduct performance experiments and exercise a modicum of control over the performance environment’s behavior. Finally, ex-

pert users are intimately acquainted with the parallel architecture and system software and want broad latitude to modify system components.

## 3 Performance Challenges

Today, the *de facto* standard for performance instrumentation and analysis is application profiling with post-mortem graphical display of performance bottlenecks. Although adequate when (a) parallel codes were written in sequential languages for homogeneous parallel systems, (b) compilers generated object code that directly reflected source code control, and (c) this code executed on a modest number of processors, this model is now woefully inappropriate for current execution environments. Today, applications are written in data parallel and object-oriented languages, compilers aggressively transform source code, and the resulting object code executes on a large, distributed collection of heterogeneous parallel systems with time varying loads.

In such an environment, performance instrumentation and analysis tools must invert compiler transformations to report measured performance in a source code context and minimize total performance data volume while still enabling users to interactively drill down to identify performance problems on remote systems. Moreover, they must accommodate execution heterogeneity and non-reproducible behavior, replacing *post-mortem* analysis with real-time analysis and optimization. Below, we describe the challenges implicit in such an approach and outline possible solutions.

### 3.1 Aggressive Optimization

Performance variability and the effort required to write explicitly parallel code have long limited the widespread use of parallel systems. Data parallel languages, like High Performance Fortran (HPF) [18] and object-parallel models like parallel C++, have been proposed to lessen the parallel programming burden. Although higher level programming models can reduce programming effort and increase code portability, they guarantee neither performance portability across parallel architectures nor scalability across problem sizes or number of processors.

Understanding performance problems and optimizing data parallel codes requires performance instrumentation and analysis tools that can relate dynamic

performance data to data parallel source code. Unfortunately, most current performance tools are targeted at the collection and presentation of program performance data when the parallelism and interprocessor communication are explicit and the program execution model closely mimics that in the source code (i.e., as is the case for message passing codes).

For data parallel languages like HPF, such instrumentation can only capture and present dynamic performance data in terms of primitive operations (e.g., communication library calls or DSM references) in the compiler-generated code. However, aggressive code restructuring and translation to a different execution model can result in executable code that differs markedly from an application developer’s model.

Moreover, complex, multidisciplinary applications often involve code written using a variety of languages and programming models and executing on multiple parallel architectures. To be effective, performance tools must not only support data parallel performance analysis, they must also support multiple architectures and provide language independent interfaces, allowing users to learn but a single set of software navigation skills.

To support source-level performance analysis of programs in data parallel languages, we believe compilers and performance tools must cooperate to integrate information about the program’s dynamic behavior with compiler knowledge of the mapping from the low-level, explicitly parallel code to the high-level source[1]. Furthermore, multilevel memory hierarchies, distributed cache coherence protocols, superscalar processors, and speculative instruction execution dictate integration of software measurements with detailed hardware performance data.

### 3.1.1 Deep Compiler Integration

To relate dynamic performance measurements of compiler-synthesized code, performance analysis tools must exploit deep knowledge, obtained from the compiler, of the translation from the high-level, data parallel source language to the low-level parallel code. Concomitantly, the compiler must synthesize instrumentation during code generation.

In an ongoing, collaborative effort with the Rice Fortran D group, we have explored performance data correlation techniques that can invert a wide range of compiler transformations, including loop interchange, code motion, and communication synthesis [1, 2]. During compilation, the Fortran D compiler emits

data on the sequence of source code transformations, tying each synthesized code fragment to a portion of the original data parallel source code.

During subsequent program execution, dynamic performance data is obtained from an instrumented version of the compiler-synthesized code. Because this dynamic data also includes pointers to the original data parallel code, a post-processing phase can compute performance metrics and map these metrics to the original data parallel source code. These metrics include both execution times and array reference data locality. The latter relies on identifying each compiler-synthesized message transmission as carrying specific array segments.

### 3.1.2 SvPablo: Portable Analysis

Building on the insights obtained from our Rice collaboration, we have developed SvPablo [8], a graphical environment for instrumenting application source code and browsing dynamic performance data. SvPablo supports performance data capture, analysis, and presentation for applications written in a variety of languages and executing on both sequential and parallel systems. In addition, SvPablo integrates data from the hardware performance counters [37] on the MIPS R10000 and SGI Origin 2000.

The SvPablo implementation relies on a single user interface for performance instrumentation and visualization. During the execution of the instrumented code, the SvPablo library captures data and computes performance metrics based on the execution dynamics of each instrumented construct on each processor. Because only statistics, rather than detailed event traces, are maintained, the SvPablo library can capture the execution behavior of codes that execute for hours or days on hundreds of processors.

Following execution, performance data from each processor is integrated, additional statistics are computed, and the resulting metrics are correlated with application source code, creating a *performance file* that is represented via the Pablo self-describing data format (SDDF) [28]. This performance file is the specification used by the SvPablo browser to display application source code and correlated performance metrics.

Using the Pablo SDDF metaformat has enabled us to develop a user interface that is both portable and language independent. Moreover, because all performance metrics are defined in SDDF, the interface is also performance metric independent, allowing us

to introduce new metrics or support new languages without change to the user interface code.

Figure 1 shows the SvPablo interface, together with code and performance data from an HPF program. As the figure suggests, the SvPablo interface supports a hierarchy of performance displays, ranging from color-coded routine profiles to detailed data on the behavior of a source code line.

Additional displays, not shown, allow one to examine detailed performance metrics for a particular code fragment on each processor. Hence, one can first identify a bottleneck, then locate its cause by interactively “drilling down” to explore increasing levels of detail.

### 3.1.3 Symbolic Performance Prediction

Although integrated compilation and performance measurement systems can relate dynamic performance data from measured executions of transformed code to the original data parallel source, they still require execution of complete programs. Often, this is too late — performance problems are due to design decisions made much earlier and can only be eliminated by large scale changes. If, during program construction, it were possible to estimate performance scalability as a function of architectural parameters, problem size, and number of processors, performance tools could further lessen the high intellectual cost of developing parallel applications by allowing application developers to understand the performance implications of data parallel code constructs.

Important questions include determining how application performance changes with variations in the parallel system configuration or application problem size, and identifying which code fragments will become the performance limiting bottlenecks as hardware or application parameters change. These predictions need not be quantitatively exact, only qualitatively exact (i.e., it is acceptable for the magnitude of prediction errors to be as high as 10–20 percent if the predictions accurately identify bottlenecks and track their movements).

Based on this observation, we have extended the Rice Fortran D95 compiler to emit performance scalability predictions [23, 22] for data parallel code. In this approach, the data parallel compiler translates data parallel code and generates a symbolic cost model for program execution time, representing the scalability of each fragment in the original data parallel source.

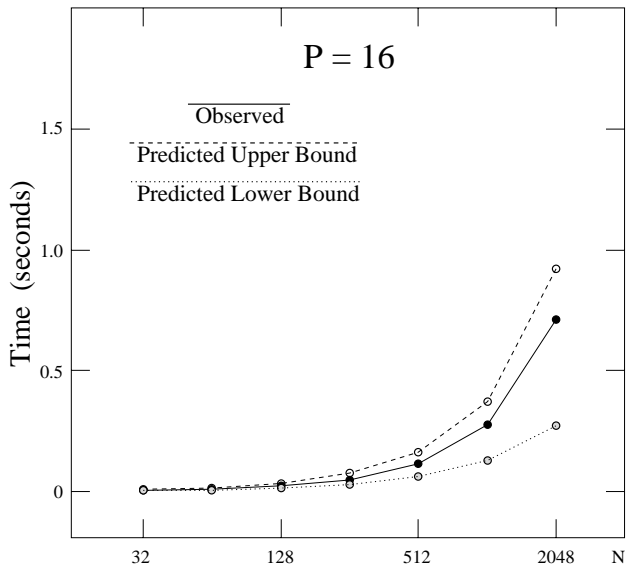


Figure 2: Symbolic Performance Prediction Example

As an example, Figure 2 shows a performance prediction as a function of problem size  $N$  for a doubly nested loop when executed on an Intel Paragon XP/S. The compiler generates upper and lower bounds for predicted execution time by considering extrema of system-dependent constants (e.g., memory references times).

### 3.1.4 Research Directions

Integrated measurement and symbolic performance prediction opens a rich, new set of opportunities for compile-time optimization. Using a symbolic manipulator, a compiler could create cost models for alternative code variants, evaluate the resulting expressions for specific numbers of processors and problem sizes, and synthesize the highest performance code variants. By augmenting and validating these symbolic models with dynamic performance data from program executions, the compiler could incrementally refine the quality of its predictions and code generation choices. In this model, compilation and performance analysis are a closed loop, with both symbolic performance predictions and quantitative measurements guiding the compilation process.

Building on these observations, we are extending our symbolic models and integrating them with the SvPablo toolkit. In addition, we are incorporating analytic and high-level simulation models for esti-

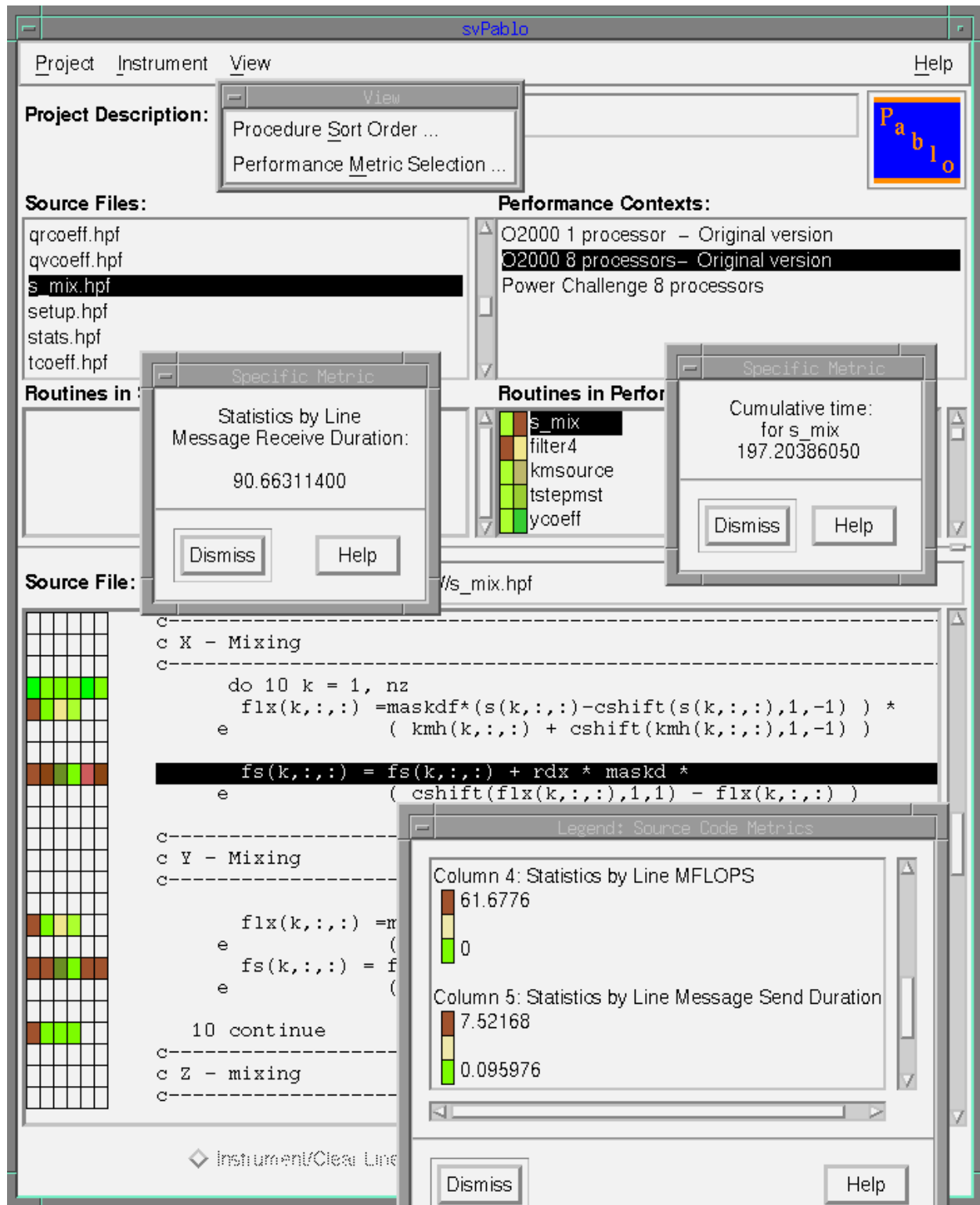


Figure 1: SvPablo Performance Display (HPF Code)

imating the behavior of caches and memory hierarchies, all based on the Polaris parallelizing compiler [3]. Our goal is to develop an integrated performance modeling, measurement, analysis, and prediction environment that will allow application and system developers to explore the performance implications of software and hardware design choices, both for extant systems and hypothetical ones.

### 3.2 Intelligent Data Reduction

As systems and applications increase in complexity and heterogeneity, identifying performance bottlenecks becomes commensurately more complex. In such cases, performance bottlenecks may lie in instruction schedulers, memory hierarchy management, wide area or local communication protocols, scheduling algorithms, or application load balance. Consequently, any performance instrumentation system must capture a large number of performance metrics if *post-mortem* analysis is to successfully identify the proximate and underlying causes of poor performance.

As we noted in §2, event traces of these metrics provide the detailed data needed to understand software component interactions, albeit potentially at great cost. Thus, one faces a conundrum: event tracing is desirable to understand detailed behavior, but the potentially large data volume, large number of performance metrics, and consequent behavioral perturbations make it impractical for large, long-running applications.

To retain the advantages of event tracing while minimizing total data volume and perturbation, one must reduce both the number of metrics needed to identify bottlenecks and the number of locations where data must be captured. To formalize this notion and to provide a basis for analysis, consider a set of  $n$  dynamic performance metrics, each measured on a set of  $P$  parallel tasks.

Conceptually, one can then view an event trace as defining a set of  $n$  dynamic performance metrics,  $m_i(t)$ , on each of  $P$  tasks

$$(m_1(t), m_2(t), \dots, m_n(t))_p \quad p \in [1..P]$$

that describe parallel system characteristics as a function of time  $t$ . Following [26], if  $R_i$  denotes the range of metric  $m_i(t)$ , we call the Cartesian product

$$M = R_1 \times R_2 \times \dots \times R_n$$

a performance metric space. Thus, the ordered  $n$ -tuples

$$(m_1(t) \in R_1; m_2(t) \in R_2; \dots; m_n(t) \in R_n) \quad (1)$$

are points in  $M(t)$ , and the event trace defines the temporal evolution of these  $P$  points in an  $n$  dimensional space.

The goal of event trace data reduction is now clear — one must reduce both the dimensionality of the metric space (i.e., reduce  $n$ ) and the number of measurement points (i.e., reduce  $P$ ). Statistical clustering and projection pursuit address the first and second problems, respectively.

#### 3.2.1 Statistical Clustering

Most programs written for parallel systems are either single program multiple data (SPMD) (e.g., using MPI), data or object parallel (e.g., using HPF or parallel C++), or functional decompositions. In the first two cases, some variation of the same code usually executes on all processors, with behavior differentiated by data-dependent control flow. In the third case, code for each function executes on different collections of processors. However, regardless of the programming model, processors executing similar code with similar data tend to form behavioral equivalence classes. In the nomenclature of (1), the processors or tasks executing similar code with comparable data trace similar trajectories in the metric space.

Dynamic statistical clustering seeks to identify clusters of processors, where the event metrics for each processor in the cluster trace similar trajectories. By periodically computing cluster membership using performance metrics from each processor, an event tracing system can capture and extract traces only from representative members of each cluster, dramatically reducing the total data volume [26].

As an example of the possible data reduction due to statistical clustering, consider an SPMD code that relies on a master task to read initialization data and allocate work to a set of  $N$  worker tasks. If the behavior of all workers is similar, clustering identifies two clusters, one with cardinality one (the master) and a second with cardinality  $N$  (the workers), yielding a total data reduction of nearly  $N$ .

Figure 3 shows an example of clustering applied to event traces from a 128 processor execution of a Hartree Fock quantum chemistry code on the Intel Paragon XP/S; see [26] for details. Clustering

groups behavior in a very small number of equivalence classes, reducing the data volume by recording trace data from only a few processors.

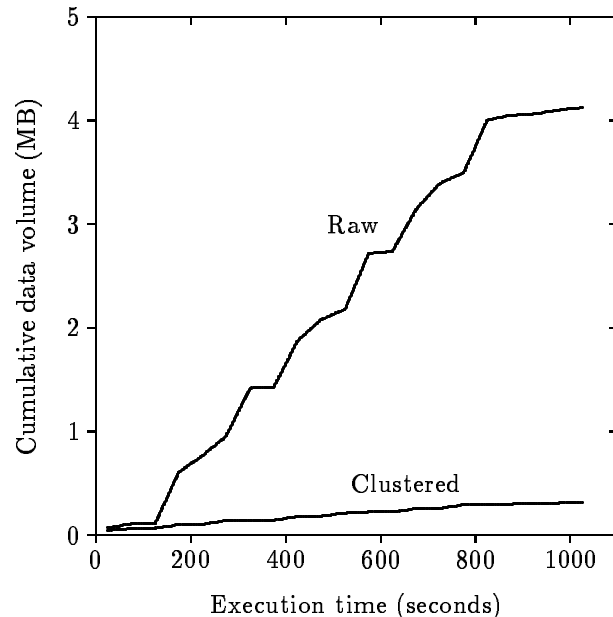


Figure 3: Statistical Clustering Data Reduction

### 3.2.2 Multidimensional Projection Pursuit

Although statistical clustering can reduce the number of processors or tasks from which event data must be recorded, it does not reduce the number of metrics or (equivalently) the dimensionality of the metric space. Even after clustering identifies a small number of processor representatives, the total data volume may remain high. As an example, when analyzing the performance of WWW servers [15], we found it necessary to capture nearly fifty metrics on request types, processor, network, and memory usage to reliably identify performance problems.

Principal component analysis and its statistical variant, projection pursuit [13], seek to identify a subset of the metrics that captures most of the statistical variation. Intuitively, at any time  $t$ , many of the  $n$  metrics are highly correlated; principal component analysis and projection pursuit identify least correlated metrics.

More formally, the  $n$  metrics (1) collectively define a vector basis for an  $n$ -dimensional coordinate system, the  $n$  orthonormal unit vectors. Projection pursuit identifies a smaller set  $k$  of orthonormal vectors, that are each linear combinations of the original

$n$  vectors. The most important (i.e., least correlated) metrics are represented as the largest components of the projection vectors

Clearly, the number of possible two or three-dimensional projections of an  $n$ -dimensional space can be large. Hence, projection pursuit minimizes an index describing the projection (e.g., in the Friedman-Tukey model [13], the index measures clot-tedness and the algorithm uses a general hill climbing technique to generate new views of the subspace).

Typically, the dimensionality  $k$  of the projection is two or three, one can visualize the projected metric space using standard visualization techniques. Moreover, because the most important (i.e., least correlated) metrics are represented as the largest components of the projection vectors, one can use the magnitude of the components to select the metrics to be recorded, reducing the total data volume. At present, we are assessing the feasibility of this approach using event traces captured from parallel systems.

### 3.2.3 Research Directions

Together, statistical clustering and projection pursuit can dramatically reduce both the number of processors and metrics from which event traces must be recorded. Implicit in such an approach is the need for real-time clustering and projection; *post-mortem* analysis requires the full trace. Indeed, in the *Autopilot* model of §3.4, real-time clustering and projection pursuit control the frequency of sensor data transmission.

Not only must the overhead for clustering and projection be sufficiently low that they do not excessively perturb the computation, they must track changes in application behavior. If clustering and projection are applied too infrequently, the selected processors and metrics may no longer be representative. Conversely, if applied too frequently, the computational overhead may excessively perturb application behavior. Hence, developing adaptive windowing algorithms for triggering clustering and projection is a key research problem.

*Autopilot* can dynamically adjust its monitoring focus on the fly, given an appropriate and automated approach to selecting interesting views. Consequently, by eliminating many uninteresting views, data volume is reduced. Within this framework, the challenge is to develop automated techniques that preserve interesting system behaviors while simultaneously reducing data volume and application per-

turbation.

### 3.3 Domain Specific I/O Analysis

Although generic instrumentation and analysis systems can identify performance bottlenecks, they rarely provide the tailored data and analysis needed to understand specific resource domains or architectures. In those areas with a sufficiently large commercial base, independent software developers (ISVs) successfully have filled this niche (e.g., with tools for network protocol analysis and tertiary storage management). However, commercial tools for specific high-performance computing domains are much less common.

Of these domains, the poor performance of parallel input/output systems has emerged as an often debilitating problem for many applications. Understanding application input/output patterns is the first step to optimizing application performance with extant file systems and to developing more effective parallel input/output file system policies.

#### 3.3.1 Scalable I/O Initiative

The Scalable I/O (SIO) Initiative is a broad-based research program that includes application and system input/output characterization, networking, file systems, and file system application programming interfaces (APIs), compiler and language support, and basic operating system services. Organized as five major working groups (performance characterization, operating systems, languages and runtime systems, applications, and software integration) that include application developers, academic researchers and vendors, the goal of the initiative is identification of parallel input/output problems and creation of software solutions.

As part of the SIO Initiative, we have extended the Pablo performance analysis toolkit to capture parallel application input/output patterns and file system responses and to compute I/O-specific performance metrics. The most significant observation from studies using this instrumentation [6, 34, 29, 30, 34, 33] is that scientific applications have input/output patterns and requirements more complex than simple stereotypes.

More specifically, our experimental data show that there is wide variation in temporal and spatial access patterns, including highly read-intensive and write-intensive phases, extremely large and extremely small

request sizes, and both sequential and highly irregular access patterns. Moreover, we have observed that the detailed spatial and temporal characteristics of input/output patterns critically affect performance. Even seemingly small variations have potentially profound performance implications. This insight motivated the development of input/output systems, described in §3.4, that can adapt to changing input/output patterns as well as creation of APIs for parallel input/output [4] that allows users and high-level libraries to specify future access patterns via hints.

#### 3.3.2 Multilevel I/O Libraries

Although the emergence of new parallel input/output APIs like the SIO API and MPI-IO promise to provide a standard interface for parallel input/output, most application developers prefer to manage data on secondary storage via high-level input/output libraries like the NCSA Hierarchical Data Format (HDF). HDF is a multi-object file format for sharing scientific data in heterogeneous, distributed environments. Via HDF routines, scientists can generate, manipulate, share, and visualize mesh and other data representations without knowledge of data storage formats or placements on parallel file systems

Because user requests via such libraries are potentially mediated by several lower level input/output libraries (e.g., implementing the MPI-IO and SIO APIs) before reaching the native file system, the potential for overhead due to information loss on library boundary crossings is high. Working in conjunction with the application and library developers we are extending our input/output characterization tools to analyze the overhead introduced by the library layers.

These instrumentation extensions target both the NCSA HDF library and the new MPI-IO standard. In both cases, the instrumentation can produce either timestamped event traces for *post-mortem* calculation of statistical summaries or compute the summaries in real-time. In either case, the summaries include the number of calls made to each high-level routine, the time spent in each routine, the time spent in low-level input/output routines to satisfy the request, and time spent in other library routines. The former allows one to explore greater detail, albeit at the potential expense of large event data volumes. Conversely, the latter trades computation perturbation for reduced event data volume.

As an example of the capabilities of this multi-level

Metric	User HDF	Internal HDF	UNIX
Count	990	265,324	43,722
Time	0.05	6.15	11.94

Table 1: HDF Routine Mix (Kleing Relativity)

input/output analysis, Table 1 shows the distribution of input/output calls for an execution of a relativity code on an SGI Power Challenge array. The table shows that a small number of application HDF library invocations generated a large number of internal HDF and UNIX input/output invocations. More detailed analysis shows that a substantial fraction of the UNIX input/output invocations were file seeks.

This “hidden cost” due to lower level input/output libraries is quite common – a modest number of high-level access to large data blocks often translates to a large number of non-sequential, smaller input/output requests. Mitigating these effects is one of the goals of intelligent library implementation.

### 3.3.3 Research Directions

Multi-level measurement of input/output libraries raises a wealth of research questions concerning library interfaces and information sharing, access pattern classification and dynamic adaptation, and tertiary storage integration. Only when critical access pattern information is shared across library boundaries can low-level input/output libraries effectively cache and prefetch data for use by higher layers.

## 3.4 Dynamic Policy Adaptation

Historically, performance analysis has been an iterative process, with performance analysts repeatedly instrumenting an application, analyzing captured performance data, and modifying the application to alleviate identified bottlenecks. With the emergence of wide area computing and computational grids, where the exact execution context may never recur, performance data captured from one execution may not highlight the critical performance issues for subsequent executions.

Moreover, complex, multidisciplinary applications with adaptive meshing, discipline-specific algorithm interactions, and real-time analysis routines, exhibit time dependent execution signatures. Experience has

shown that matching resource policies to time varying application behavior can greatly increase application performance (e.g., tuning input/output policies for caching and prefetching can reduce application input/output costs by an order of magnitude [20]).

Based on these observations, we believe the historical model of *post-mortem* performance analysis and optimization must be subsumed by a new model of real-time analysis and adaptive optimization. Such adaptive systems respond to both changes in execution context and resource availability and to time varying application resource demands, dynamically adjusting resource policies to maximize achieved performance.

### 3.4.1 Autopilot Adaptive Control System

Development of closed loop adaptive performance analysis systems minimally requires three components:

- intelligent *decision procedures* that determine how and when the system should adjust resource allocation policies and system parameters,
- distributed *performance sensors* that collect performance data for decision procedures, and
- resource *policy actuators* that implement change parameters and policies in response to decisions.

Below, we describe the design of *Autopilot*, a closed loop performance measurement and adaptive control system that includes these components.

**Flexible Decision Procedures.** Common decision procedure mechanisms like decision tables or trees rely on a consistent division of the parameter space, assigning policies and policy parameters to each division. Likewise, controllers derived using classic control theory are based on a rigorous mathematical analysis of the underlying system. However, the dynamic behavior of heterogeneous distributed systems and irregular multidisciplinary applications are too poorly understood to be amenable to either of these control techniques.

In contrast, fuzzy logic targets precisely the attributes of the resource management problem that challenge classic techniques [14], namely conflicting goals and poorly understood optimization spaces. *Autopilot* builds on this observation by coupling a configurable fuzzy logic rule base for distributed decision making with wide area performance sensors

and policy control actuators. The rule base embodies common sense rules for resource management (e.g., aggressively prefetch file blocks for small, sequential requests).

**Performance Sensors.** *Autopilot's* performance instrumentation is based on a set of distributed sensors that extract information from the component tasks of executing applications. In addition to the standard requirements of low overhead and minimal perturbation, sensors must accommodate wide area computation where some of the tasks may execute on remote systems with different architectures than the clients receiving the data.

The *Autopilot* sensor requirements are driven by a desire to minimize application perturbation, reduce communication frequency and volume, and maximize flexibility. Based on these constraints, *Autopilot* sensors support local data buffering and reduction, dynamic connection to remote tasks, and dynamic activation and removal.

Local data buffering and reduction (e.g., computing sliding window averages from measurements) trades computation perturbation at the site of data capture for reduced communication frequency and data volume. Because data transmission is often the most expensive part of data collection, local data reduction can dramatically decrease measurement perturbation.

Because the metrics needed to identify performance bottlenecks and optimize applications change with application behavior, decision mechanisms must be able to dynamically activate and deactivate sensors in response to changing conditions. Finally, because source code is not always available, particularly for proprietary software libraries, one of the *Autopilot* sensor design goals is support for dynamic instrumentation, allowing sensors to be inserted or removed from a program executable.

**Policy Actuators.** Resource policy actuators form the final component of the *Autopilot* triumvirate, proving the mechanism to change application parameters or resource policies during execution. Driven by the output of decision procedures, actuators have many of the same properties as sensors, including local computation and distributed activation.

### 3.4.2 Adaptive Parallel File Systems

As our extensive analysis of input/output dynamics [6, 35, 32] has shown, the parallel input/output patterns in emerging applications are both irregular and dynamic. Because the interactions between these applications and the file system software change during and across application executions [19], it is difficult or impossible to determine a globally optimal input/output configuration or to statically configure runtime systems and resource management policies for parallel input/output. Hence, parallel input/output optimization provides an excellent test of an adaptive, closed loop control system like *Autopilot*.

Based on these experiences, we have designed and started the implementation of a second generation portable parallel file system, PPFS II, with real-time, adaptive policy control. PPFS II is designed to work atop either parallel systems or PC and workstation clusters, providing a flexible testbed for high-performance input/output experiments.

To explore automatic, qualitative classification of resource use, we have developed a suite of trained artificial neural networks (ANNs) [19] and hidden Markov models (HMMs) [20] that are implemented as *Autopilot* sensors. ANNs can efficiently classify access streams in real-time. In contrast, HMMs build a probabilistic model of the access pattern using prior execution training. This generality allows HMMs to classify arbitrary access patterns.

In addition to automatic behavioral classification techniques, PPFS also includes a flexible set of fuzzy logic rules that can intelligently select file system policies based on input/output resource demands and supplies. In this context, we are developing a set of adaptive, dynamic policies to enhance the parallel input/output system performance, including adaptive striping and dynamic storage redundancy, that can trade storage space for increased input/output bandwidth and balance conflicting resource demands within and across applications.

### 3.4.3 Research Directions

Earlier, we argued that deep integration of performance tools and compilation systems was necessary to map dynamic performance data to application source code. Because an increasing fraction of applications rely on proprietary software libraries where source code is not available, comprehensive, dynamic instrumentation must include both compiler-synthesized measurements and object code instru-

mentation. Although some binary rewriting tools can capture instruction frequencies, relating this data to source code is difficult. Current practice is restricted to object code instrumentation at procedure boundaries, and even this presumes limited procedure inlining. To enable comprehensive instrumentation, compilers must record sufficient metadata for later object code instrumentation even when source code is not available.

To complement comprehensive instrumentation, intelligent data acquisition, perhaps based on the statistical clustering and projection pursuit techniques of §3.2, are needed to dynamically adjust the types and volume of performance data. Any adaptive resource management system must intelligently enable and disable sensors to minimize data volume while maintaining enough information to correctly detect critical decision points.

We also see the role of performance analyst as evolving from diagnosing individual performance problems to creating rule bases and making critical instrumentation decisions. Currently, there are no formal methods or useful tools that support creation and validation of rule sets for adaptive control. Fuzzy logic seems promising, but new methods, such as neural networks and genetic algorithms, can also be used to learn control rules and select membership functions [24]

Finally, while neural networks and hidden Markov models have proven useful for input/output characterization [20], other application resource characterizations remain to be explored. Concurrently, better techniques for identifying and exploiting global characterizations must be developed.

### 3.5 Direct Manipulation

Although statistical clustering, projection pursuit, and adaptive control techniques can all reduce performance data volume and lessen the intellectual burden of performance optimization, the application developer or performance analyst can gain insight only by exploring the measured data. With rapidly rising software complexity and system heterogeneity, this has become an increasingly odious and tedious task.

Simply put, performance visualization and analysis tools have not kept pace with the rapid rise in application and system complexity. In contrast to the acceptance of immersive virtual environments for analysis of complex scientific data, simple static and dynamic workstation graphics remain the *de facto* stan-

dard for performance analysis. For performance analysts to effectively visualize and optimize the behavior of tens of distributed parallel systems connected by high-bandwidth networks, accessing distributed secondary and tertiary storage systems, and collectively containing thousands of processors, new performance visualization systems must more fully exploit human sensory capabilities.

Not only has the number of factors that can affect performance increased, effective performance tuning of distributed applications often requires cooperative exploration by a group of physically dispersed domain experts (e.g., networking, storage, scheduling, and architecture). Consequently, their analysis tools must reflect the collaborative, often asynchronous nature of their work.

#### 3.5.1 *Virtue* Performance Data Immersion

Although there are many possible designs for an immersive performance environment, we believe all successful designs will share the majority of the following features:

- *hierarchical views* that can represent parallel software components and their interactions,
- *attribute controls* for interactive modification and adjustment of software attributes and parameters,
- *direct manipulation tools* for modifying system behavior during execution,
- *multimedia annotation software* for asynchronous collaboration, and
- *actualization interfaces* that connect the virtual environment with the external world, realizing the effects of modifications.

These beliefs are buttressed by our experience with virtual environment visualization of WWW traffic and parallel input/output [31].

Below, we describe the design of *Virtue*, a collaborative environment for immersive performance analysis that embodies these design components. At present, a prototype of *Virtue* is operational, supporting three-dimensional visualization in the CAVE [7], but much work remains to complete implementation of all design components.

**Hierarchy and Abstraction.** As we noted in §2, the key limitation of extant performance visualization tools is their lack of scalability. To address this problem, *Virtue* realizes abstract data as a hierarchy of embedded, three-dimensional graphs. Within the virtual environment, one can interactively expand and contract subgraphs to explore details at a particular site.

In this model, distributed computations are initially represented as geographic graphs whose vertices represent specific computation sites and whose edges represent intersite communication. By mapping performance metrics to visual attributes like graph vertices and edges, the geographic graph provides an overview of the computation.

By selecting a specific site, one can expand the associated subgraph to see additional detail on the computation at that site. Thus, this model of hierarchy allows one to move from a geographic view, through a single parallel system view, to a task graph or procedure call view on a specific processor.

**Direct Manipulation Tools.** Although a few performance visualization systems accept real-time data streams and display software behavior during execution, most rely on *post-mortem* data. In consequence, performance data analysis and visualization is a largely passive process — one cannot change parameters or policies of the application or underlying system and see their effects except by modifying and re-executing the code.

*Virtue*'s design is predicated on the belief that users can best optimize performance by interacting with virtual objects (i.e., the hierarchical graphs representing system components) just as they would objects in the physical world. This direct manipulation model makes the user a participant in the performance experiment rather than merely an observer.

*Virtue* graphs include manipulators (e.g., 3-D sliders and buttons) that are accessible via a tactile feedback data glove. By touching graph components, one can expand or contract subgraphs and change the behavior of executing software (e.g., toggling a caching policy or changing a network packet size).

**Distributed Collaboration.** Just as computations increasingly involve distributed access to remote resources, application software developers and performance analysts are themselves often distributed. For these groups to collaborate effectively, performance visualization systems must be *reflective*, al-

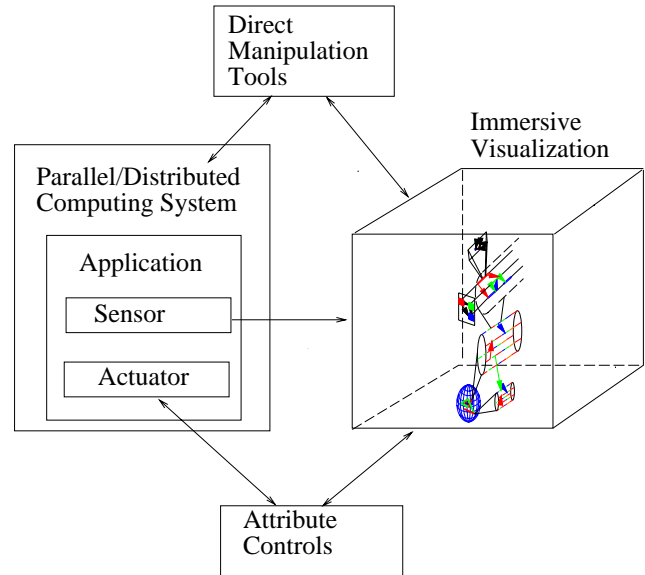


Figure 4: *Virtue* and *Autopilot* Coupling

lowing distributed groups to share visualizations and insights both synchronously and asynchronously. For synchronous collaboration, live audio and video from all participants are displayed within *Virtue*, allowing them to cooperatively explore the graph hierarchy.

In contrast, asynchronous collaboration mandates sharing across both time and space, by annotating interesting phenomenon with notes and insights for later reference and exploration by collaborators. *Virtue* includes a multimedia annotation system that allows users to select objects in a *Virtue* graph, and attach an audio/video annotation via voice commands. When subsequent participants reactivate the visualization, these annotations, denoted by icons attached to graph objects, can be selected and replayed.

**Remote Actualization.** As we noted earlier, direct manipulation allows users to change *Virtue* graph representations and parameters. To realize the effects of these changes, the virtual environment must be coupled to the system from which data is being captured. In *Virtue* this coupling or actualization is via *Autopilot* actuators.

Just as *Autopilot* sensors provide the data sources needed for real-time visualization in *Virtue*, actuators provide the interface for realizing virtual environment manipulations. As Figure 4 suggests, *Virtue* replaces closed loop software control via fuzzy logic decision procedures with interactive, adaptive control.

### 3.5.2 Research Directions

Immersive, three-dimensional visualization of software structure and execution dynamics opens a wide range of research problems in psychometrics and usability, manipulation interfaces and controls, and training of adaptive system software. In particular, understanding which mappings of performance data to visual (e.g., shape, color, placement, size, and opacity), sonic, and tactile attributes provide the most insight is a key open problem.

Statistical techniques like clustering and projection pursuit are natural complements to immersive visualization, reducing the possible domain of exploration and focusing attention on statistically relevant metrics. However, dynamic metric selection must be constrained by the need to maintain visual continuity for human interaction.

Quite clearly, the time scale for interactive analysis and resource policy optimization precludes certain types of microscale optimization (e.g., selecting synchronization primitive policies on a millisecond time scale). However, it effectively supports macroscale optimization (e.g., file caching policies on a minute or hour time scale). Intelligently integrating software control and adaptation with interactive optimization will require balancing occasionally conflicting goals.

Finally, learning algorithms for software rule bases could potentially generalize from examples of interactive optimization. Alternatively, users could interactively adjust the parameters of extant rule bases. In such a meta-optimization model, the virtual environment is the substrate for configuring closed loop adaptive controls systems.

## 4 Related Work

A large number of *a priori* performance prediction and *a posteriori* performance measurement and analysis tools have been developed, targeting both sequential and parallel systems — far more than can be summarized here. Notable examples include P<sup>3</sup>T [10] for performance prediction, together with Paradyn [25] and AIMS [36] for performance measurement. Each has exposed key research issues in performance measurement and analysis.

Similarly, several systems have been built that support application behavior steering (i.e., guiding a computation toward interesting phenomena), though there have been fewer efforts to interactively steer or adaptively control application performance. Notable

examples include Leblanc's [16] creation of an adaptive real-time system for robotic control that consists of a multiprocessor executing a group of adaptive cognitive tasks and Schwan *et al*'s [9] development of adaptive control mechanisms based on a sensor/actuator model.

Performance data visualization has a long and storied history, though the explosion of interest in visualization of parallel system behavior can be traced to Seecube [5], Pablo [28] and ParaGraph [12]. Virtual environments for performance analysis [27] are less common, though emerging immersive systems for distributed collaboration [17] target many of the same problems.

## 5 Conclusions

As parallel computing evolves from homogeneous parallel platforms and applications dominated by single algorithms to heterogeneous collections of parallel systems and multidisciplinary applications, performance measurement, analysis, and optimization problems continue to increase. With this rise in complexity, performance measurement systems must continue to evolve, integrating performance data from a wider range of sources and shifting from *post-mortem* optimization to real-time adaptive management. Concurrently, analysis and visualization systems must support the reality of distributed collaboration and hierarchical visualization.

## Acknowledgments

In addition to the contributions of the authors, this work draws on the insights and hard work of many past and present members of the Pablo research group, most notably Thomas Kwan, Oleg Nickolayev, Phil Roth, Luis Tavera, Will Scullin, and Evgenia Smirni.

## References

- [1] ADVE, V., MELLOR-CRUMMEY, J., WANG, J.-C., AND REED, D. Integrating Compilation and Performance Analysis for Data-Parallel Programs. In *Proceedings of Supercomputing'95* (November 1995).
- [2] ADVE, V. S., MELLOR-CRUMMEY, J., ANDERSON, M., KENNEDY, K., WANG, J., AND REED,

- D. A. Integrating Compilation and Performance Analysis for Data-Parallel Programs. In *Proceedings of the Workshop on Debugging and Performance Tuning for Parallel Computing Systems*, M. L. Simmons, A. H. Hayes, D. A. Reed, and J. Brown, Eds. IEEE Computer Society Press, 1994.
- [3] BLUME, W., DOALLO, R., EIGENMANN, R., GROUT, J., HOEFLINGER, J., LAWRENCE, T., LEE, J., PADUA, D., PAK, Y., POTTINGER, W., RAUCHWERGER, L., AND TU, P. Parallel Programming with Polaris. *IEEE Computer* (Dec. 1996).
- [4] CORBETT, P. F., PROST, J.-P., DEMETRIOU, C., GIBSON, G., RIEDEL, E., ZELENKA, J., CHEN, Y., FELTEN, E., LI, K., HARTMAN, J., PETERSON, L., BERSHAD, B., WOLMAN, A., AND AYDT, R. Proposal for a Common Parallel System Programming Interface Version 1.0. <http://www.cs.arizona.edu/sio/api1.0.ps>, Nov. 1996.
- [5] COUCH, A. *Graphical Representations of Program Performance on Hypercube Message-Passing Multiprocessors*. PhD thesis, Tufts University, Department of Computer Science, 1988.
- [6] CRANDALL, P. E., AYDT, R. A., CHIEN, A. A., AND REED, D. A. Characterization of a Suite of Input/Output Intensive Applications. In *Proceedings of Supercomputing '95* (Dec. 1995).
- [7] CRUZ-NEIRA, C., D.J.SANDIN, AND DEFANTI, T. Surround-Screen Projection-Based Virtual Reality: The Design and Implementation of the CAVE. In *SIGGRAPH '93 Proceedings* (Aug. 1993), Association for Computing Machinery.
- [8] DEROSE, L., ZHANG, Y., AND AYDT, R. Sv-pablo: A Multi-Language Performance Analysis System. In submitted for publication (Mar. 1998).
- [9] EISENHAEUER, G., GU, W., SCHWAN, K., AND MALLAVARUPU, N. Falcon — Toward Interactive Parallel Programs: the Online Steering of a Molecular Dynamic Program. In *Proceedings of the Third International Symposium on High-Performance Distributed Computing* (Aug. 1994).
- [10] FAHRINGER, T. Estimating and Optimizing Performance for Parallel programs. *IEEE Computer* 28, 11 (November 1995), 47–56.
- [11] GRAHAM, S., KESSLER, P., AND MCKUSICK, M. gprof: A Call Graph Execution Profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* (Boston, MA, June 1982), Association for Computing Machinery, pp. 120–126.
- [12] HEATH, M. T., AND ETHERIDGE, J. A. Visualizing the Performance of Parallel Programs. *IEEE Software* (Sept. 1991), 29–39.
- [13] HURLEY, C., AND BUJA, A. Analyzing High-dimensional Data with Motion Graphics. *SIAM Journal of Scientific and Statistical Computing* 11, 6 (Nov. 1990), 1193–1211.
- [14] KASABOV, N. K. *Foundations of Neural Networks, Fuzzy Systems, and Knowledge Engineering*. The MIT Press, 1996.
- [15] KWAN, T. T., MCGRATH, R. E., AND REED, D. A. NCSA's World Wide Web Server: Design and Performance. *IEEE Computer* (Nov. 1995), 68–74.
- [16] LEBLANC, T. J., AND MARKATOS, E. P. Operating System Support for Adaptive Real-time Systems. In *Proceedings of the Seventh IEEE Workshop on Real-Time Operating Systems and Software* (May 1990), pp. 1–10.
- [17] LEIGH, J., JOHNSON, A., AND DEFANTI, T. CAVERN: Distributed Architecture for Supporting Persistence and Interoperability in Collaborative Virtual Environments. *Virtual Reality Research, Development and Applications* 2, 2 (Dec. 1997), 217–237.
- [18] LOVEMAN, D. B. High Performance Fortran. *IEEE Parallel & Distributed Technology* 1, 1 (February 1993), 25–42.
- [19] MADHYASTHA, T. M., AND REED, D. A. Exploiting Global Input/Output Access Pattern Classification. In *Proceedings of SC '97: High Performance Computing and Networking* (San Jose, Nov. 1997), IEEE Computer Society Press.
- [20] MADHYASTHA, T. M., AND REED, D. A. Input/Output Access Pattern Classification Using Hidden Markov Models. In *Proceedings of the*

- Fifth Workshop on Input/Output in Parallel and Distributed Systems* (San Jose, CA, Nov. 1997), ACM Press, pp. 57-67.
- [21] MALONY, A. D., REED, D. A., ARENDT, J. W., AYDT, R. A., GRABAS, D., AND TOTTY, B. K. An Integrated Performance Data Collection Analysis, and Visualization System. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications* (Monterey, CA, Mar. 1989), Association for Computing Machinery.
  - [22] MENDES, C. L. *Performance Scalability Prediction on Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, May 1997.
  - [23] MENDES, C. L., WANG, J.-C., AND REED, D. A. Automatic Performance Prediction and Scalability Analysis for Data Parallel Programs. In *Proceedings of the CRPC Workshop on Data Layout and Performance Prediction* (Houston, April 1995).
  - [24] MEREDITH, D. L., KARR, C. L., AND KAMUR, K. K. The Use of Genetic Algorithms in the Design of Fuzzy Logic Controllers. *3rd Workshop on Neural Networks WNN'92* (1992), 549-545.
  - [25] MILLER, B. P., CALLAGHAN, M. D., CARGILLE, J. M., HOLLINGSWORTH, J. K., IRVIN, R. B., KARAVANIC, K. L., KUNCHITHAPADAM, K., AND NEWHALL, T. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer* 28, 11 (November 1995), 37-46.
  - [26] NICKOLAYEV, O. Y., ROTH, P. C., AND REED, D. A. Real-time Statistical Clustering for Event Trace Reduction. *International Journal of Supercomputer Applications and High Performance Computing* (1997).
  - [27] OSAWA, N. An Enhanced 3-D Animation Tool for Performance Tuning of Parallel Programs based on Dynamic Models. In *Proceedings of the Symposium on Parallel and Distributed Tools* (July 1998).
  - [28] REED, D. A., AYDT, R. A., NOE, R. J., ROTH, P. C., SHIELDS, K. A., SCHWARTZ, B., AND TAVERA, L. F. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference* (1993), A. Skjellum, Ed., IEEE Computer Society.
  - [29] REED, D. A., ELFORD, C. L., MADHYASTHA, T., SCULLIN, W. H., AYDT, R. A., AND SMIRNI, E. I/O, Performance Analysis, and Performance Data Immersion. In *Proceedings of MASCOTS '96* (Feb. 1996), pp. 1-12.
  - [30] REED, D. A., GARDNER, M. J., AND SMIRNI, E. Performance Visualization: 2-D, 3-D, and Beyond. In *Proceedings of the IEEE International Computer Performance and Dependability Symposium* (Sept. 1996).
  - [31] REED, D. A., SHIELDS, K. A., TAVERA, L. F., SCULLIN, W. H., AND ELFORD, C. L. Virtual Reality and Parallel Systems Performance Analysis. *IEEE Computer* (Nov. 1995), 57-67.
  - [32] SIMITCI, H., AND REED, D. A. A Comparison of Logical and Physical Parallel I/O Patterns. *International Journal of Supercomputer Applications and High Performance Computing* (to appear 1998).
  - [33] SMIRNI, E., ELFORD, C. L., AND REED, D. A. Performance Modeling of a Parallel I/O System: An Application Driven Approach. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing* (Mar. 1997).
  - [34] SMIRNI, E., AND REED, D. A. I/O Requirements of Scientific Applications: An Evolutionary View. In *Proceedings of the Fifth IEEE International Symposium on High-Performance Distributed Computing* (Aug. 1996), pp. 49-59.
  - [35] SMIRNI, E., AND REED, D. A. Workload Characterization of Input/Output Intensive Parallel Applications. In *Proceedings of the 9th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (June 1997).
  - [36] YAN, J. C., SARUKKAI, S. R., AND MEHRA, P. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. *Software Practice & Experience* 25, 4 (April 1995), 429-461.
  - [37] ZAGHA, M., LARSON, B., TURNER, S., AND ITZKOWITZ, M. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings of Supercomputing'96* (November 1996).